Using Streaming SIMD Extensions in a Motion Estimation Algorithm for MPEG Encoding

Version 1.2

01/99

Order Number: 243652-002

Table of Contents

Revision History

Revision	Revision History	Date
1.2	FCS update.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. V.Bhaskaran, K.Konstantinides. Image and video compression standards. Algorithms and Architectures.
2. Mitchell Joan.L., Pennenbaker W, Fogg C, LeGall D.J (1996).MPEG video compression standard.pp.21-31,283-307.
3. MPEG standard, Coding of Moving Pictures, ISO/IEC DIS 11172.
4. Feig E., and S. Winograd, (1992). Fast Algorithms for Discrete Cosine Transform, IEEE Trans. Signal Proc., 40, pp. 2174-2193.
5. MMX™ technology application notes: Using MMX™ Instructions to Compute the Absolute Difference in Motion Estimation, Using MMX™ Instructions to Implement Data Alignment, Using MMX™ Instructions to Implement Optimized Motion Compensation for MPEG1 Video Playback; at `http://developer.intel.com/drg/mmx/appnotes/`

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating point single-instruction, multiple-data (SIMD) instructions, as well as new integer SIMD instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio.

This application note illustrates how to use the Streaming SIMD Extensions and MMX™ technology instructions to perform motion estimation (ME) for the MPEG Encoder. The application note includes examples of code that exploit the new instructions. In particular, it uses the packed sum of absolute differences (`psadbw`) instruction to implement a fast motion-estimation error function.

This application note covers these main points:

- Explains some of the basic principles of motion estimation.
- Describes a fast implementation of the motion estimation error calculation, using a sum of absolute differences.
- Provides estimates of performance improvements using the new instructions.

2 Motion Estimation

Motion estimation (ME) is a video compression technique performed during video stream encoding. ME benefits situations in which:

- Most of an object's characteristics, such as shape and orientation, stay the same from frame to frame.
- Only the object's position within the frame changes.

The ME module in most encoders is very computation-intensive, so these kernels are typically optimized as much as possible.

2.1 Motion Vectors

ME calculates the motion vector of a 16 by 16 pixel region, know as a macroblock. The motion vector is the relative displacement of the macroblock from one frame to another.

Assume that (x, y) is the position of upper-left corner of a macroblock in the current frame at time (t). If the best matching macroblock in a reference frame is located at (x+u, y+v), the motion vector associated with the macroblock at location (x, y) is the vector (x+u, y+v). In relative terms, this vector is expressed as (u, v). The vector is a forward motion vector if the reference frame is at an earlier time (t-n). The vector is a backward motion vector if the reference frame is at a later time, (t+n).

2.1.1 Picture Types

During the encoding process, picture types are defined for different frames. Within a single frame, each macroblock can be coded with a different picture type. The number of motion vectors needed per macroblock depends of the macroblock's picture type.

- Intra pictures (I-frames) are coded without reference to other frames. An I-type macroblock does not require motion vectors.

- Predicted pictures (P-frames) are coded using motion-compensated prediction from a past I-frame or P-frame. Within a P-frame, a macroblock can be coded as a P-type or an I-type macroblock. A P-type macroblock requires one motion vector per macroblock. The I-type coding is used if the motion vector found during ME cannot be used.
- Bidirectionally-predicted pictures (B-frames) are coded using both past I- or P-frames and future I- or P-frames. Within a B-frame, a macroblock can be coded as an I-type, P-type, or B-type macroblock. A B-type macroblock uses an interpolation of a forward motion vector and a backward motion vector. B-frames typically require two motion vectors per macroblock. B-frames are never used as a reference frames for prediction.

2.2 Motion Estimation Process

The motion estimation process is the search for the best matching macroblock in the reference frame. Rather than search the entire reference frame, typical algorithms restrict the search to a [-p, p] region around the (x, y) location of the macroblock in the current frame.

The “best match” may be determined in many ways. One common, computationally-simple method is to find the macroblock for which the absolute difference between the pixels in the two frames is minimal. That is, the “best matching” macroblock is the block for which the Mean Absolute Error function, MAE (i, j), is minimized.

MAE(i ,j) is defined as follows:

$$MAE(i,j) = \frac{1}{MN}\sum_{k=0}^{M-1}\sum_{l=0}^{N-1}\left|C(x+k,y+l) - R(x+i+k,y+j+l)\right|$$

where:

C(x+k, y+l)	denotes the pixels of the macroblock at location (x, y) in the current frame. This macroblock is referred to as the "estimated" macroblock.
R(x+i+k, y+j+l)	denotes the pixels of the macroblock in the reference frame. This macroblock is referred to as the "reference" macroblock.
i, j	are defined in the search region -p<=i<=p, -p<=j<=p.
N, M	are the dimensions of the macroblock.

The motion vector for the estimated macroblock at (x, y) is the vector (i, j) that minimizes the function, MAE(i ,j).

2.2.1 Block Matching Algorithms

Block matching algorithms are search techniques that find the value of (i, j) that minimizes the Mean Absolute Error function, MAE(i ,j). Two possible algorithms are:

- full-search algorithm
- two dimensional (2D) logarithmic search algorithm

The full-search algorithm computes MAE(i, j) at each location (i, j) in the search region, [-p, p]. Although this algorithm guarantees that the minimum MAE(i, j) is found, the computation takes more time than is practical for most encoders.

The 2D logarithmic search algorithm is similar to a binary search. First, the algorithm computes MAE(i, j) at 9 locations within the search region: (0, 0) and 8 points located along the rectangle [-p/2, p/2]. Using the "best match" location as the new starting point (0, 0), the algorithm computes MAE(i, j) at 9 more locations: the new starting point and 8 points located along the rectangle [-p/4, p/4]. This process continues until the search region cannot be divided further. The best match at this stage defines the motion vector.

The complexity of the 2D logarithmic search algorithm is only 3.3% of the complexity of the full search algorithm[1]. However, if MAE(i, j) does not increase monotonically, this algorithm may find a local minimum for MAE(i, j), rather than the global minimum.

For simplicity's sake, the code in this application note implements the 2D logarithmic search algorithm for P macroblocks only, and for full pel (picture elements, or pixels) movements only. The logarithmic search was chosen to demonstrate some effects of less-than-optimal cache usage.

2.2.2 Minimum Error Threshold

The error, MAE(i, j), is small when the best matching macroblock is nearly the same as the estimated macroblock. In fact, motion estimation should be used only for macroblocks whose value of MAE(i, j) is less than a predefined threshold. If MAE(i, j) is greater than the threshold, a macroblock should be encoded as an I-type macroblock. The threshold value is not defined by the MPEG standard, so some degree of flexibility is possible.

This application note only finds the motion vector; it does not implement the threshold comparison to determine whether to use the motion vector for encoding.

3 Implementing the Motion Estimation Algorithm

The file, `\samples\mot_est\MEstub.c`, contains the full C code implementation of two search algorithms:

- `PfullSearch`, implements the full search algorithm.
- `PlogarithmicSearch`, implements the logarithmic search algorithm.

Both `PfullSearch` and `PlogarithmicSearch` call a function that calculates the mean absolute error for motion estimation, MAE(i, j), which is the sum of absolute differences between a macroblock in the reference frame and the macroblock in the current frame. The best matching block is the block that minimizes this error.

The error function is implemented in three ways:

- `MotionErrorC`, calculates the error using C code.
- `MotionErrorXMM`, calculates the error using Streaming SIMD Extensions.
- `MotionErrorMMX`, calculates the error using MMX instructions.

To see how the error calculation works, look at the C-code example listed in Example 1. The algorithm loops over a 16-by-16 pixel macroblock, computing the absolute value of the difference between

[1] Motion estimation for 720 x 480 pixel frames at 30 fps in a [-15,15] search region requires 29.89 GOPS for the full search algorithm and only 1.02 GOPS using a 2D logarithmic search algorithm. See [1].

corresponding pairs of pixels in the estimated macroblock (current frame) and the reference macroblock, and summing the results in `diff`.

The code contains a "fast out" option that saves loop iterations by comparing the error value accumulated after each row (`diff`) with the error value for the current, best matching macroblock (`bestdif`). If `diff` > `bestdif`, the remaining loop iterations are skipped. This option is commented out due to tradeoff considerations related to the Streaming SIMD Extensions implementation discussed in Section 6.

```
#define ABS(x) (((x)<0)?-(x):(x))
typedef unsigned char uint8;

uint8* currblock;  /* pointer to rows of 16x16 pixel macroblock in current frame.*/
uint8* refblock;   /* pointer to rows of 16x16 pixel macroblock in reference */
                   /*frame.*/

for ( y = 0; y < 16; y++ ) {
     /* update pointer to the start or next row*/
   refblock = &(frame[(fy+y)*FRAME_WIDTH + fx]);
     /* update pointer to the start or next row*/
   currblock = &block[y*BLOCK_SIZE];

   for ( x = 0; x < 16; x++ ) {
      localdiff = refblock[x]-currblock[x];
      diff += ABS(localdiff);
   }

   //  if ( diff > bestdif ) {
   //     return diff;
   //  }
}

return diff;
```

Example 1: C Code of 16x16 Motion Estimation Error Calculation

3.1 Error Function Parameters

Each implementation of the error function takes these parameters:

- A pointer to the estimated macroblock in the current frame, which is a 16x16 matrix of 8-bit elements. This matrix should be aligned on a 16-byte boundary. This pointer is located in the `ebx` register.
- A pointer to the reference frame, which is a matrix of 8-bit elements. The size of this matrix depends on the picture size. This pointer should be aligned on a 16-byte boundary.
- The two coordinates of the reference macroblock inside the reference frame. These coordinates determine whether the data in the reference macroblock is 16-byte aligned. A pointer to the upper

left pixel of the reference macroblock is calculated based on these two coordinates, and stored in the `edx` register.

One parameter that is not used in the current routine is the best difference found prior to the current macroblock. If the absolute difference accumulated across rows is more than the difference for the current best match, the iterations for the remaining part of the macroblock can be skipped. However, a branch instruction is needed for the comparison. Since the `psadbw` instruction is so efficient, the cost of additional iterations is less that the cost of the branch instruction, due to potential mispredictions.

The `MotionErrorXMM` function returns the sum of the absolute differences of the pixels in the two macroblocks. Some definitions used in the error function are:

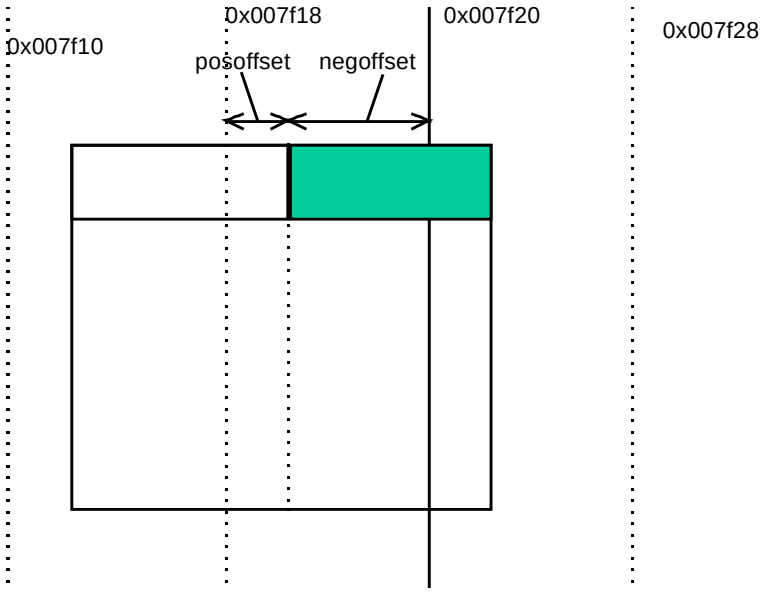
- `FRAME_WIDTH`, the width of the reference frame, in bytes.
- `BLOCK_SIZE`, the width of the reference macroblock, in bytes.
- `END_BLOCK`, a pointer to the last pixel in the macroblock.
- `POSOFFSET`, the number of bits from next highest 8-byte aligned address to the left border of reference macroblock.
- NEGOFFSET, the value `64 – POSOFFSET`.
- 

Figure 1: Macroblock Offset from 8-byte Aligned Address

3.2 Error Function in Streaming SIMD Extensions

Example 2 shows one loop of the Streaming SIMD Extensions implementation of the error function calculation. This loop calculates the sum of absolute differences for one row of a macroblock. The loop has two main stages:

1. Reading data from the reference macroblock (reference frame) into MMX technology registers.
2. Summing the absolute differences between the reference macroblock data in the MMX technology registers with the estimated macroblock (current frame) data in memory.

```
;first loop for simple read

 movq    mm1,[edx]       ;read 1st 8 pixels from reference block.
                         ;edx – pointer to reference block (unaligned)
 movq    mm2,8[edx]      ;read next 8 pixels from reference block.
 psadbw  mm1,[ebx]       ;calculate absolute difference of pairs of 1st 8 pixels
                         ;ebx – pointer to estimated block (aligned)
 psadbw  mm2,8[ebx]      ;calculate absolute difference of pairs of next 8 pixels
 paddw   mm6, mm1        ;add to buffer for final sum of absolute differences
 paddw   mm7, mm2        ;add to buffer for final sum of absolute differences
```

Example 2: Core of Error Calculation Using Streaming SIMD Extensions

In the actual code, the loop is unrolled four times, so each loop iteration calculates the sum of absolute differences for 4 rows of the macroblock.

3.2.1 Reading Data from the Reference Block

The error function in the code using Streaming SIMD Extensions uses two `movq` instructions to read one row of the reference macroblock (16 pixels) into two MMX technology registers. The reference macroblock may be aligned or unaligned, depending on its relative location from estimated macroblock. The estimated macroblock is always aligned, due to its location within the aligned estimated frame.

This application note compares two techniques for reading data from the reference macroblock:

- a simple data read that does not try to prevent data cache unit (DCU) splits; these splits are memory references that span two cache lines.
- a more complex data read that prevents DCU splits.

3.2.1.1 Simple Data Read

When the `SPLIT_ALIGN` flag in the `MotionErrorXMM` error function is set to `OFF`, the function uses a simple routine to read data from the reference macroblock. For each row in the macroblock, two `movq` instructions load 8 pixels each. Even though the reference frame itself is aligned, the reference macroblock may be unaligned within the frame.

Whenever the left pixel of each row of the macroblock (16 bytes per row) is located in the upper half of a cache line (meaning bytes 16-31), one of the `movq` instructions crosses a data cache unit (DCU) border, causing a cache line split. This occurs in ¼ of all memory accesses.

3.2.1.2 Preventing DCU Splits

When the `SPLIT_ALIGN` flag in the `MotionErrorXMM` error function is set to `ON`, the function executes code that prevents DCU splits when reading data. For this technique, the left pixel of each row of the reference macroblock should be same distance from the next higher 32-byte aligned address. By choosing an appropriate value for `FRAME_WIDTH`, you can guarantee that each row of the macroblock is equally aligned or unaligned.

If the left pixel of each row of the macroblock is one of the lower 17 bytes in a cache line (0-16), no special treatment is needed and data can be read using two `movq` instructions, as described in Section 6.2.

If the left pixel of each row of the macroblock lies within the 17-23 bytes of a cache line, a DCU split occurs in the Y2 and Y4, 8x8 blocks, as shown in Figure 2. If the left pixel of each row of the macroblock lies within the 24-31 bytes of a cache line, the DCU split occurs in the Y1 and Y3, 8x8 blocks. In these cases, for each 8-pixel row, the routine loads 16 aligned pixels to two registers and extracts the correct 8 pixels from the registers using SHIFT and OR instructions [3]. The values of `POSOFFSET` and `NEGOFFSET` are used to determine the appropriate number of bits for shifting data.

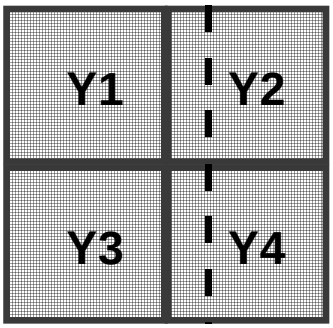


Figure 2: Macroblock Offset from 8-byte Aligned Address

This approach requires two conditional branches to determine which way to read the data. The potential for branch misprediction can be avoided by determining in advance which routine to call. For example, when p=16 for a logarithmic search algorithm, the first 9 search positions are aligned. In this case, no penalty occurs for using the simple data reading routine.

3.2.2 Calculating the Absolute Difference

The error function in the code using Streaming SIMD Extensions uses two `psadbw` instructions to calculate the absolute difference between the pixels in one row of the reference macroblock and the corresponding row of the estimated macroblock. The `psadbw` instruction computes the absolute value of the difference of unsigned, packed bytes for the two source operands. These differences are summed to produce a word result in the lower word of the destination, while the upper words of the destination are cleared.

The `mm6` and `mm7` registers are used as buffers to sum the absolute difference results for each row.

The Assembly file, `\samples\mot_est\MEXMM.asm`, contains the function, `MotionErrorXMM`, which calculates the mean absolute error for motion estimation using Streaming SIMD Extensions. The error function calculates the sum of the absolute differences between the pixel values of two 16-by-16 macroblocks: a reference macroblock in the reference frame and the estimated macroblock in the current frame. The estimated macroblock is always 16-byte aligned, but the reference macroblock may be aligned or unaligned, depending on its relative location from estimated macroblock.

4 Performance

The performance of the motion estimation application can be improved through the use of several features of the Streaming SIMD Extensions, as described in the following sections.

4.1 Gains/Improvements

The Streaming SIMD Extensions improve performance for several reasons:

- Implementing a sum of absolute differences for 16 pixels in MMX technology code requires about 20 MMX instructions, including packed subtract, packed addition, logical, and unpack instructions. Implementing the same calculation with Streaming SIMD Extensions requires only two `psadbw` instructions.
- Any improvement in the error calculation has a large impact, since the search algorithm performs so many block-by-block comparisons.
- The absolute difference calculation does not contain any branch instructions, which reduces potential delays due to branch mispredictions.
- Unrolling the loop four times save on loop overhead, meaning fewer instructions are executed.

4.2 Considerations

This section describes special considerations for improving performance.

4.2.1 DCU Split Handle

This application note investigates the impact of data cache unit (DCU) splits in the motion error function. Due to the nature of the search algorithm, the next matching macroblock may or may not be aligned on a 32-byte boundary. For half of the unaligned macroblocks, half of the data reads are unaligned. Thus, the code incurs a significant number of DCU split penalties when executing on a Pentium® III processor.

An alternative approach is to read data from aligned addresses only, and use a combination of `SHIFT` and `OR` instructions to extract the appropriate data. Assuming that motion vectors are uniform, DCU splits occur in only 1/4 of the data reads. Thus, if the same `SHIFT-OR` routine is used for all motion vectors, the extra instructions needed to extract the data are unnecessary in 3/4 of the data reads.

Another approach is to use two different routines, one with `SHIFT-OR` instructions and one without. The function jumps to the appropriate routine based on the X coordinate of the motion vector. This solution prevents DCU splits entirely, but adds a conditional branch with the risk of branch misprediction.

Current measurements show only a 1.05x speedup for the code that handles DCU splits. This result occurs because of increased instruction count and branch mispredictions.

4.2.2. Prefetch

The synthetic data used in this app note causes memory conditions to be such that the performance of the code example can not be improved by using the `prefetch` instruction. In a real application, the `prefetch` instruction can be used to improve performance.

One way to use the `prefetch` instruction is to prefetch the data of the estimated block. Since precise block position in the estimated frame is known, `prefetch` can be used once every two blocks to prefetch sixteen 32-byte cache lines for the two next blocks. To avoid prefetching more than once, the `prefetch` instruction must be placed outside of the loop of motion vector search.

It is difficult to know what data to prefetch for the next reference block, since its position varies depending on previous searches, and on the nature of the search algorithm. Vtune can help analyze cache conditions and identify further opportunities for optimization.

5 Conclusion

The motion estimation module in most encoders is very computation-intensive, due to the large number block-by-block comparisons. Streaming SIMD Extensions provide a fast way of performing the fundamental motion-error calculation using the `psadbw` instruction to compute the absolute difference of unsigned, packed bytes.

6 Code Examples

The `\samples\MotionEst\` subdirectory contains the C and assembly code files discussed in this application note:

- `MEstub.c` contains the C code implementation of two search algorithms, `PfullSearch` and `PlogarithmicSearch`, and the motion error function, `MotionErrorC`. There is also an associated header file named `mestub.h`.
- `MEMMX.asm` contains the MMX technology code implementation of the motion error function, `MotionErrorMMX`.
- `MEXMM.asm` contains the Streaming SIMD Extensions code implementation of the motion error function, `MotionErrorXMM`.